# TIME REVERSIBILITY: A MATHEMATICAL TOOL FOR CREATING ARBITRARY GENERALIZED SWAP-WITH-PARENT SELF-ORGANIZING LISTS

John Oommen and Juan Dong

TR-97-08 MAY 1997

School of Computer Science, Carleton University
Ottawa, Canada, KIS 5B6

# Time reversibility: a mathematical tool for creating arbitrary generalized swap-with-parent self-organizing lists *

John Oommen
Sch. of Comp. Sci.
Carleton University
Ottawa, Canada
oommen@scs.carleton.ca

Juan Dong
Sch. of Comp. Sci.
Carleton University
Ottawa, Canada
dong@turing.scs.carleton.ca

**Abstract**— *Self organizing linear search algorithms have been in the literature for almost 30 years, and numerous schemes have been proposed during that time. Among all the previous algorithms, the move-to-front rule and the transposition rule are the most extensively analyzed schemes. Recently we have proposed and thoroughly analyzed a new scheme, the swap-with-parent rule, which views the list as a heap structure with no ordering constraints between parents and their children [15]. From the analyses of the transposition rule and the swap-with-parent rule, it can be seen that the fundamental property of the corresponding Markov chain being time reversible greatly simplifies the analysis of the algorithm. In this paper, we shall show the existence of a class of time reversible Markov chains resulting from performing swaps on "implicit" tree structures (called ss_trees) which generalize and extend the results concerning the transposition heuristic and the swap-with-parent heuristic.*

*This paper introduces a generalization of the transposition rule and the swap-with-parent rule - the* swap-with-parent-in-an-ss_tree *heuristic and its modification - the* move-to-parent-in-an-ss_tree *heuristic. Detailed expressions for the asymptotic probabilities and the asymptotic search cost of the scheme have been derived.*

**Keywords**— *Self organizing lists, the move-to-front rule, the transposition rule, the swap-with-parent rule, time reversible Markov chains, ss_trees, swap-with-parent-in-an-ss_tree, move-to-parent-in-an-ss_tree.*

## 1 Introduction

This paper presents a novel approach to designing arbitrary, user-defined self-organizing sequential search algorithms by viewing the list as an implicit tree.

Suppose we are given a set of records $R_1, R_2, \ldots, R_n$ which are in an arbitrary order $\pi$, so that $R_i$ is in position $\pi(i)$ for $1 \leq i \leq n$. At every instant of time one of these records $R_i$ is accessed. We do so by examining each record of the list starting from the first record until $R_i$ is found. This search costs $\pi(i)$ units of time to perform.

We assume that each record $R_i$ is accessed with an unknown probability $s_i$ and that the accesses are made independently. The expected search cost (average search length) for an ordering

*Partially supported by the National Sciences and Engineering Research Council (NSERC) of Canada

of the list $\pi$ is thus

$$cost(\pi) = \sum_{1 \le i \le n} s_i \pi(i). \tag{1}$$

To minimize the access cost, it is desirable that the records are ordered in the descending order of their access probabilities. We shall refer to this kind of list ordering as a perfect ordering or optimal ordering and refer to a file ordered in this way as a completely organized file. However the access probabilities are seldom known *a priori* in practice, and so the list can not be arranged in the *optimal ordering* in advance. In this situation we need an algorithm which dynamically rearranges the list and gradually transforms it to a less costly ordering.

*A self-organizing sequential search list* is a linear search list in which the order of the records may be altered each time a sequential search occurs so that after sufficiently many accesses, it tends to be in the optimal ordering with high probability where the most frequently accessed record is at the front of the list and the rest of the list is recursively ordered in the same manner. The most useful and common type of alteration is to move the accessed record forward one or more positions in the list. In this way more frequently accessed records move towards the front of the list so that fewer comparisons are needed on subsequent accesses.

This problem of having a file organize itself has been studied extensively. Various reviews on self organizing strategies have been published over the years. We refer the reader to the comprehensive surveys of the papers and results in the area [5, 6, 8] and the surveys on the applications [2, 8, 19]. Rather than describe the various strategies in any detail, we shall briefly highlight some of their main features.

An intuitive scheme for reordering a list to a, hopefully, less costly ordering is to keep a counter of accesses for each record, and maintain the records in the descending order of their access frequencies. However, as Knuth remarks ( [12] p398 ), this counter scheme is undesirable as it requires extra memory space which could perhaps be better used by employing nonsequential search techniques.

The first memory-free self-organizing scheme proposed by McCabe in 1965 [13] is the *move-to-front* rule. In this scheme, each time a record is accessed, it is moved to the front of the list, and all the records before the accessed record are shifted back one position. The asymptotic search cost of the scheme in terms of the average number of probes required to find a record in the given list (after it has reached a *steady state* where many further reorderings on the list are not expected to increase or decrease the search cost significantly) is

$$1 + 2 \sum_{1 \le i < j \le n} \sum \frac{s_i s_j}{s_i + s_j}. \tag{2}$$

Many other researchers [5, 6, 7, 8] have also extensively studied the $MTF$ rule and various properties of its limiting convergence characteristics are available in the literature.

McCabe [13] introduced another memory-free scheme called the *transposition* rule. In this rule, the accessed record is moved one position closer to the front of the list by interchanging it with its preceding record unless it is at the front of the list. This was later proved by Rivest [19] to have lower expected search cost per access than the move-to-front rule, and he conjectured that the transposition rule is optimal. This conjecture was further strengthened by the result by Bitner [4] that for some special distributions the transposition rule is optimal over all rules. However Anderson *et al.* [1] found a counterexample to this conjecture by deriving a rule that is better than the transposition rule for a specific distribution. Bitner [3, 4] later showed that while the transposition rule is asymptotically more efficient, the move-to-front rule converges more quickly and proposed a hybrid of these two rules that attempts to incorporate the best features

of both. This simple hybrid rule initially uses the move-to-front algorithm until its steady state is approached, and then switches to the transposition rule. However the difficulty of deciding when to switch is still a major problem.

Rivest [19] proposed a compromise between the relative extremes of the move-to-front rule and the transposition rule, namely the *move-ahead-k* heuristic where the accessed record is moved $k$-positions forward towards the front of the list unless it is in the first $k$ positions, in which case it is moved to the front. This is a generalization of the transposition rule and the move-to-front rule as transposition is *move-ahead-1* and move-to-front is *move-ahead-n*. However no absolute analyses is available for this scheme.

Tenenbaum and Nemes [21] suggested a generalization of the move-to-front rule and the transposition rule, the *POS(k)* rule. The POS($k$) rule moves the accessed record to position $k$ of the list if it is in positions $k+1$ to $N$, or it transposes it with its preceding record if it is in positions 2 to $k$. If it is the first in the list, it is left unchanged. Note that *POS(1)* is the move-to-front, whereas *POS($n-1$)* is the transposition strategy.

Notice that all the algorithms described above alter the list on a single access basis. We shall call such an algorithm a *reordering algorithm* or *permutation algorithm*.

McCabe [13] considered reorganizing the list only once every $k$ accesses to reduce the time spent reordering the list. It is to be used in conjunction with permutation algorithms. In this scheme, a counter is needed for each record to store the value of $k$. A record is moved forward (according to the permutation algorithm chosen) only if it has been accessed $k$ times, not necessarily in a row, and then the counter for *that record* is reset. Bitner [4] also studied this rule and analyzed its performance when used with the move-to-front rule. Bitner also suggested a modification to it, the *wait c, move and clear* rule. After a record has been accessed $c$ times, not necessarily in a row, it is moved forward and the counter for *every record* is reset.

Kan and Ross [9] and Gonnet *et al.* [6] proposed the *k-in-a-row* heuristics, where a record is moved forward only after it is accessed $k$ times in a row. Gonnet *et al.* [6] proved that $(k+1)$-*in-a-row* is superior to *k-in-a-row* and suggested a minor modification called the *k-in-a-batch* heuristic, where accesses are grouped into batches of size $k$, and a record will be moved if it is accessed $k$ times in a batch. They proved that the batched-$k$ rule is better than the $k$-in-a-row rule when in combination with either the move-to-front rule or the transposition rule. Note that these rules all require extra space for storing the values of $k$.

Since the Markov chains representing the schemes described above are ergodic, the list can be in any of its $n!$ configurations. As opposed to ergodic representations, Markovian behavior can also be absorbing [11, 20]. In an absorbing Markov chain, the chain converges to one of a (finite) set of absorbing barriers. Oommen and Hansen [16] introduced two absorbing list organizing schemes, the *bounded memory stochastic move-to-front* scheme and the *stochastic move-to-rear* scheme. In both algorithms, the move operation is performed stochastically in such a way that ultimately no more move operations are performed. However it is shown that the first scheme is never better than the deterministic move-to-front algorithm. For the second scheme, they showed that the probability of convergence to the optimal ordering could be made as close to unity as desired. Oommen *et al.* [17] later proposed two deterministic absorbing schemes, both of which perform move-to-rear operation. One is the *deterministic linear-space move-to-rear* scheme, the other is the *deterministic constant-space move-to-rear* scheme. The former moves the accessed record to the *rear* of the list if it has been accessed $k$ times. The scheme is asymptotically optimal, the probability of being absorbed into the optimal ordering can be made as close to unity as desired. The second scheme moves the accessed record to the *rear* of the list if it has been accessed $k$ *consecutive* times. It is proven to be expedient and its variant was proven to be optimal [18]. Again all the above probabilistic schemes require extra memory.

Notice that there is a common drawback to all the algorithms described above, that is, they all fail to consider the size of the list when reorganizing it. The two extremes are the well-known move-to-front rule and transposition rule. Oommen and Dong recently proposed two memory-free algorithms [5, 15], both of which take into account the size of the list when reorganizing it. The first algorithm is called *swap-with-parent* (SWP) and the second is called *move-to-parent* (MTP).

Under the *swap-with-parent* heuristic, the accessed record gets exchanged with its "parent" (considering the list as a heap structure with no ordering constraints between parents and their children), and all the other records stay unchanged. Instead of swapping the accessed record with its "parent", the *move-to-parent* heuristic moves the accessed record to its parent's position and shifts the parent and all the records between the accessed record and its parent back one position. They showed that under the swap-with-parent heuristic, the Markov chain representing the scheme is *time reversible*. Using this property, they further derived various expression for the asymptotic probabilities and for the average search cost of the scheme. Although the scheme is no better than the transposition rule in terms of the average search cost per access, they conjecture that it costs no more than the move-to-front rule and its convergence is intermediate to the move-to-front rule and the transposition rule. It takes as few as $\lfloor \lg n \rfloor$ steps for a frequently accessed record to be moved from the back of the list to the front (root) of the list as opposed to $n$ accesses needed for the transposition rule, and as few as $\lfloor \lg n \rfloor$ steps for a less frequently accessed record to be moved from the front of the list to the back of the list as opposed to $n$ accesses needed for the transposition rule. The appendix of this paper contains various expressions which we have derived for this scheme.

For the performance of the move-to-parent rule, empirical comparison shows that it lies between the move-to-front heuristic and the transposition heuristic, and that it is better than the swap-with-parent rule.

Among all the previous algorithms, the move-to-front rule, the transposition rule and the swap-with-parent rule are the three algorithms which have been most thoroughly analyzed. We notice from the analyses of the transposition rule and the swap-with-parent rule, that the fundamental property of the corresponding Markov chain being time reversible makes the analysis of the algorithm possible and greatly simplified.

This paper shows the existence of a class of time reversible Markov chains resulting from performing swaps on "implicit" tree structures (ss_trees) which generalize and extend the results concerning the transposition heuristic and the swap-with-parent heuristic. It, thereafter, introduces a generalization of the transposition rule and the swap-with-parent rule - the *swap-with-parent-in-an-ss_tree* (SWPSST) heuristic and its modification - the *move-to-parent-in-an-ss_tree* (MTPSST) heuristic.

The performance of both of the schemes depends heavily on the type of the underlying ss_tree used. In both of the schemes, we believe that in terms of accuracy, the best-case scenario happens if the underlying ss_tree is the unary tree, i.e., the tree with one branch, in which case the scheme is actually the transposition rule. This heuristic, however, is the most sluggish scheme in terms of convergence. The worst-case scenario occurs if the underlying ss_tree is an $(n-1)$-branch tree. Generally speaking, the 'bushier' the ss_tree is, the more the algorithms costs, and the faster the convergence rate is. We conjecture that SWPSST heuristic costs less than the move-to-front heuristic and that the MTPSST heuristic is even better. Further analyses for both of the schemes are needed. This indeed gives rise to numerous open problems.

As in the literature concerning the theory of self organizing data structures, we shall use the terms *algorithm, rule, scheme* and *heuristic* interchangeably.

We will first give a brief description of time reversible Markov chains.

## 2 Time Reversible Markov Chains

Some Markov chains have the property that the process behaves in just the same way regardless of whether time is measured forwards or backwards. Kelly [10] made an analogy saying that, "if we take a film of such a process and then run the film backwards the resulting process will be statistically indistinguishable from the original process." This property is described formally in the following definition.

**Definition 2.1** *A stochastic process $X(t)$ is time reversible if a sequence of states $(X(t_1), X(t_2), \ldots, X(t_n))$ has the same distribution as the reversed sequence $(X(t_n), X(t_{n-1}), \ldots, X(t_1))$ for all $t_1, t_2, \ldots, t_n$.* □

Consider a stationary ergodic Markov chain (that is, a Markov chain that has been in operation for a long time) having transition probabilities $M_{st}$ and stationary probabilities $P\{\pi_s\}$. Suppose that starting at some time we trace the sequence of states going backwards in time. That is, starting at time $t$, consider the sequence of states $X_t$, $X_{t-1}$, $X_{t-2}$, $\ldots$ $X_0$. It turns out that this sequence of states is itself a Markov chain with transition probabilities $Q_{st} = (P\{\pi_t\}/P\{\pi_s\}) * M_{ts}$. If $Q_{st} = M_{st}$ for all $s, t$, then the Markov chain is said to be *time reversible*. Note that the condition for time reversibility, namely $Q_{st} = M_{st}$, can also be expressed as

$$P\{\pi_s\} M_{st} = P\{\pi_t\} M_{ts} \qquad \text{for all } s \neq t. \tag{3}$$

The condition in the above equation can be stated as, for all states $s$ and $t$, the rate at which the process goes from $s$ to $t$ (namely $P\{\pi_s\} M_{st}$) is equal to the rate at which the process goes from $t$ to $s$ (namely $P\{\pi_t\} M_{ts}$). It is worth noting that this is an obvious necessary condition for time reversibility since a transition from $s$ to $t$ going backward in time is equivalent to a transition from $t$ to $s$ going forward in time. Thus, if $\pi_m = s$ and $\pi_{m-1} = t$, then a transition from $s$ to $t$ is observed if we are looking backward, and one from $t$ to $s$ if we are looking forward in time.

The following theorem adapted from Ross ([20]) gives the necessary and sufficient condition for a finite ergodic Markov chain to be time reversible. The proof of the theorem can be found in [20] p.143.

**Theorem 2.1** *A finite ergodic Markov chain for which $M_{st} = 0$ whenever $M_{ts} = 0$ is time reversible if and only if starting in state $s$, any path back to $s$ has the same probability as the reversed path. That is, if*

$$M_{s,s_1} M_{s_1,s_2} \ldots M_{s_k,s} = M_{s,s_k} M_{s_k,s_{k-1}} \ldots M_{s_1,s}$$

*for all states $s, s_1, \ldots, s_k$.* □

Using the above theorem, we shall show, in the next section, that any tree structure associated with finite stationary Markov process is time reversible.

## 3 A Class of Time Reversible Markov Chains

Inspired from the result that the Markov chain representing the swap-with-parent heuristic is time reversible [5, 15], we now follow the avenue of thought that a Markov chain resulting from

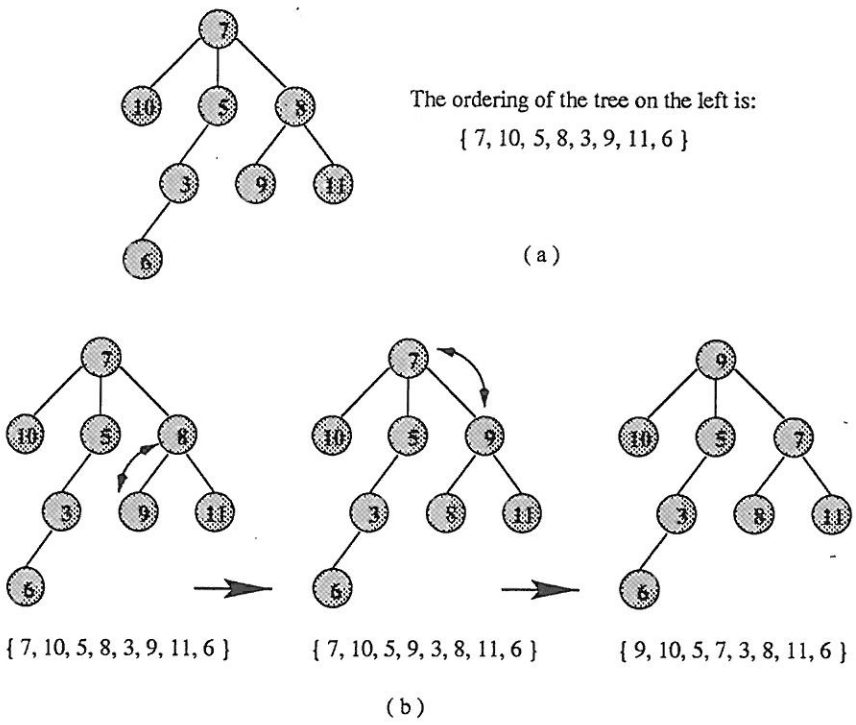



Figure 1: Tree orderings and transformations on repeated accesses of record "9".

the "swapping" operation on any tree structure is time reversible. In fact, this result is not totally new. Kelly ([10], p9) proved the following lemma.

**Lemma** (Adapted from Kelly [10].) *If the graph $G$ associated with a stationary Markov process is a tree, then the process is time reversible.* □

Although Kelly reported this result, he did not demonstrate how to associate a tree with a stationary Markov chain. In this section, we shall give a formal definition for such a tree structure in the self-organizing list application domain and prove the corresponding theorem regarding its time reversibility.

Let $T$ be an arbitrary tree structure with $n$ nodes. Let the ordering of the tree be the sequential order of the nodes in the tree (see Figure 1 (a)). At every time instant one of the node in the tree is accessed, with probability $s_i$, and it is swapped with its parent. Therefore every swap transforms the tree from one ordering to another, and there are total of $n!$ tree orderings. Let the ordering of the tree be the states of a system. Since it is possible to reach any state from any other by a sequence of one or more transformations (see Figure 1 (b)), the transformations form an irreducible ergodic Markov chain with the state at any time being the tree ordering at that time. The transition probabilities $M_{st}$ for the Markov chain are determined by the access probabilities $s_i$ of the nodes and the swapping operation used. Each $M_{st}$ will either be zero or one of the $s_i$'s. The eigenvector of $M = \{ M_{st} \mid 1 \leq s \leq n! \text{ and } 1 \leq t \leq n! \}$ corresponding to the eigenvalue unity gives the stationary probabilities $P\{\pi_s\}$ of the states $\pi_s$ $(1 \leq s \leq n!)$ of the system. We shall call such a process a *tree transformation process.*

Using this tree transformation process we now introduce a generalization of the transposition and the swap-with-parent heuristic. We shall call it the *swap-with-parent-in-an-ss_tree*

(SWPSST) heuristic invoked by what we call a *sequential search trees* (ss_trees). Before we go into any details, we shall first define the underlying tree structure.

### 3.1 Sequential Search Trees

Oommen and Dong introduced the concept of sequential heap or s_heap where we view a list as a heap without the heap property [5, 15] (see also the Appendix of this paper). Here we shall define a similar tree structure, the *sequential search tree* or *ss_tree.*

**Definition 3.1** *A sequential search tree is an unordered list viewed as a tree. The ordering of the tree is the ordering of the list, and the position of the parent of each node is fully defined in terms of the position of the node itself. Trivially, the parent of the root is the root itself.*

The structure of an ss_tree is not a tree, but an unordered list. It is only a tree on a conceptual level ("implicit") like the s_heap we defined in [5, 15]. Therefore there is no lexicographic ordering among the nodes. We therefore cannot perform any tree operations such as *rotation* etc.. Figure 2 shows various ss_trees given in the following examples.

**Example 1.**
*A unary ss_tree* is a list viewed as a tree with only one branch. The parent of each node is its previous node. The list operated by the transposition rule is a unary ss_tree, and the operation performed on the list is the generalized "swap with parent" of the ss_tree. Figure 2 (a) shows a list with 5 elements viewed as a unary ss_tree.

**Example 2.**
*A binary ss_tree* is the s_heap introduced in [5, 15]. In this case, if node $R_i$ has index $\pi(i)$ in the list, then the parent of $R_i$ has index $\lfloor(\pi(i)/2)\rfloor$ in the list, and the children of node $R_i$ have index $2\pi(i)$ and $(2\pi(i)+1)$. A list with 9 elements viewed as a binary ss_tree is shown in Figure 2 (b).

**Example 3.**
We now define a *k-branch ss_tree* where $k < n$. The root node has $k$ children and every other node has one child. Therefore if node $R_i$ has index $\pi(i) > k$ in the list, then its parent has index $\pi(i) - k$. Conversely if $\pi(i) < k$, the parent of node $R_i$ is the root. Similarly the children of the root are the nodes with index $\{2, \ldots, k+1\}$, and the only child of the node with index $j > 1$ has index $(j + k)$. A list with 13 elements viewed as a 3-branch ss_tree is shown in Figure 2 (c).

**Example 4.**
A *leftist* binary ss_tree is a tree in which the root of the tree has two children and every left child of a node has two children (except at the leaf level) but every right child of a node has no children. Therefore if node $R_i$ has index $\pi(i) > 3$ and $\pi(i)$ is an even number, then the parent of node $R_i$ has index $(\pi(i) - 2)$. However, if $\pi(i)$ is an odd number, the parent of node $R_i$ has index $(\pi(i) - 3)$. If $\pi(i) \leq 3$, the parent of node $R_i$ is the root node. Figure 2 (d1) shows a leftist binary ss_tree with 7 elements and $k = 2$.

Similarly we can define the *leftist k-ary ss_trees* Figure 2 (d2) and (d3) show the ss_trees with $k = 3$ and $k = 4$ respectively.

**Example 5.**
A *rightist* binary ss_tree is symmetric to the leftist tree. The root of the tree has two children and every right child of a node has two children (except at the leaf level) but every left child of a node has no children. Therefore if node $R_i$ has index $\pi(i) > 3$ and $\pi(i)$ is an even number, then the parent of node $R_i$ has index $(\pi(i) - 1)$. However, if $\pi(i)$ is an odd number, the parent of node

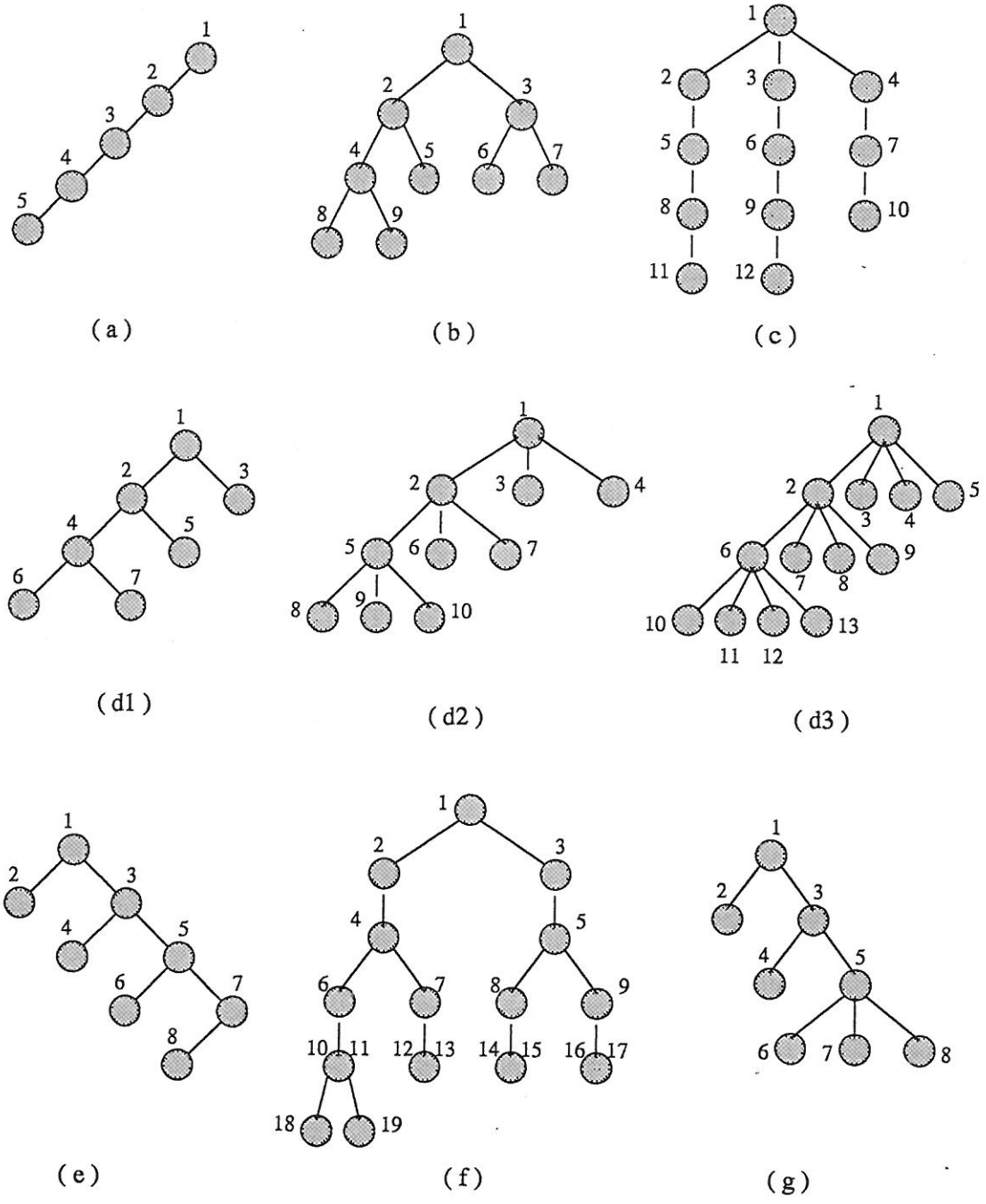



Figure 2: Various ss_trees. The number beside each node represents the index of the node in the list.

$R_i$ has index $(\pi(i) - 2)$. If $\pi(i) \leq 3$, then the parent of node $R_i$ is the root node. Figure 2 (e) shows a rightist binary ss_tree with 8 elements.

Similarly we can define the *rightist k-ary ss_trees.*

**Example 6.**
Arbitrary ss_trees are trees where there is no pre-defined ordering but only the parent/child ordering for specific positions. Examples of these are shown in Figure 2 (f) and (g). In these case, the parent/child relationship must be tabulated.

We shall show that if we apply the tree transformation process described in the previous section to an ss_tree associated with a list, then the resulting Markov chain is time reversible. That is to say, if we have an ss_tree, and the only operation allowed on the ss_tree is that of swapping a node with its parent, the resulting Markov chain is time reversible. This leads us to a new class of heuristics for self-organizing sequential search.

### 3.2 The Swap-with-Parent-in-an-SS_Tree Heuristic

Suppose we are given a list of records $\{R_1, R_2, \ldots, R_n\}$ and an ss_tree associated with it. At every time instant one of these records $R_i$ is accessed with probability $s_i$. Whenever a record is accessed, it is swapped with its parent in the corresponding ss_tree. We call this list reordering rule the *swap-with-parent-in-an-ss_tree* (SWPSST) rule.

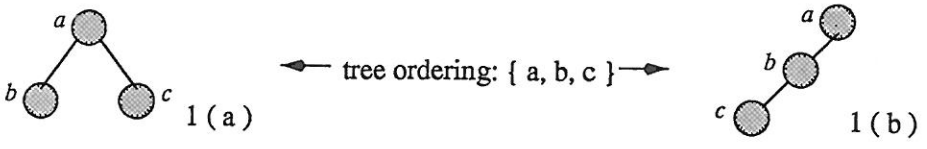
First of all, note that the SWPSST rule is memory-free. Since, by the definition of the ss_trees, the index of the parent of each record is fully defined by the index of the child record, in the actual implementation, the only thing we need is to have a temporary variable to store the position of the parent of the accessed record during the search. The scheme is a generalization of the transposition heuristic and the swap-with-parent heuristic as the transposition heuristic is swap-with-parent-in-an-*unary*-ss_tree and the swap-with-parent heuristic is swap-with-parent-in-a-*binary*-ss_tree. As we shall see later, many results that hold for the transposition rule and the swap-with-parent rule can be generalized for the swap-with-parent-in-an-ss_tree rule. The most straightforward one is the time reversibility shown in the theorem below.

**Theorem 3.1** *The Markov chain which results from using the swap-with-parent-in-an-ss_tree heuristic is time reversible.*

**Proof:** We will carry the proof by induction on the number of elements, $n$, in the tree, and we will use the 'box' (sub-chain) notation which we developed for the proof of the time reversibility of the transposition rule (see [5, 14]).

*1. Base case* $n = 3$.

For any tree with three elements, the tree can be in only one of the two structures shown below:




If the tree structure is 1(a), then the scenario is the same as the swap-with-parent rule on an s_heap of three elements. Similarly, if the tree structure is 1(b), then the scenario is the same

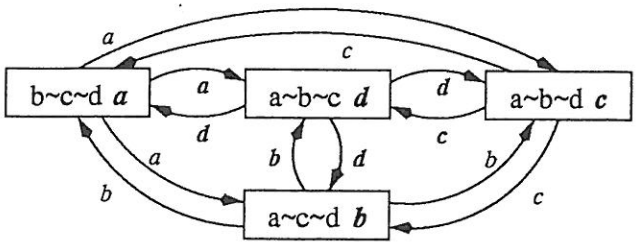



Figure 3: Simplified state diagram with four elements per state using the SWPSST rule with an arbitrary ss_tree.

as the transposition rule. Since in both cases, the Markov chain is time reversible, therefore, the Markov chain with three elements is time reversible in all cases.

2. *Case* $n = 4$.

Before we start the induction hypothesis, we shall look at the case of $n = 4$.

Using Theorem 2.1, to prove that a Markov chain is time reversible, it is sufficient to prove that starting in any state $\pi_s$, any path back to $\pi_s$ has the same probability as the reversed path.

The Markov chain for $n = 3$, has six states resulting from permuting the three elements $a, b, c$. Regardless of the tree structure, using the 'box' notation, the Markov chain can always be denoted by a~b~c. After adding one more element 'd' to the list, the simplified state diagram for the 'four element' Markov chain can be represented by Figure 3.

Since each sub-chain represented by a box in Figure 3 is exactly the same as the Markov chain with three elements (e.g., the chain represented by a~b~c **d** is actually the chain with the first three elements, a~b~c), transitions within each 'box' preserve time reversibility. Also, since all four sub-chains are symmetric, it will be sufficient to prove that if we start in any one of the sub-chains, say, a~b~c **d**, and in any state $\pi_s$ from *that* sub-chain (a~b~c **d**), any path back to $\pi_s$ has the same probability as the reversed path.

Without loss of generality, let $\pi_s = \{a, b, c, d\}$, and assume that the parent of '$d$' is '$a$', then any path back to $\pi_s$ from inside the sub-chain will have the same probability as the reversed path since we have shown that the Markov chain consisting of states entirely within this sub-chain is time reversible. So it remains to show that any path back to $\pi_s$ which goes *outside* the box has the same probability as the reversed path. We know that to reach any states outside the sub-chain, element **d** must be accessed. After $d$ is accessed, we are in state $\{d, b, c, a\}$ (since '$a$' is the parent of '$d$') in sub-chain b~c~d **a**, and so to get back to state $\pi_s$, we have to get back to sub-chain a~b~c **d**. Since all sub-chains are symmetrical, we only need to consider the path back to a~b~c **d** directly from b~c~d **a**, which, in turn, means that the last state we will visit inside b~c~d **a** right before we get back to a~b~c **d** must be a state where '$d$' is the parent of '$a$'. Since $\{d, b, c, a\}$ has already been visited when we first entered the sub-chain b~c~d **a**, we will not be allowed to use it again. There is thus only one choice - we must get back from state $\{d, c, b, a\}$. Obviously, the shortest path is then

$$\{a, b, c, d\} \rightleftharpoons \{d, b, c, a\} \rightleftharpoons \{d, c, b, a\} \rightleftharpoons \{a, c, b, d\} \rightleftharpoons \{a, b, c, d\}\,.$$

The product of the transition probabilities in the forward direction is $s_d s_c s_a s_b$, and in the reverse direction is $s_c s_d s_b s_a$. . The equality in Theorem 2.1 holds.

The argument holds for a larger path which goes around visiting more states inside $\boxed{\text{b}\sim\text{c}\sim\text{d a}}$. This is because the large path inside $\boxed{\text{b}\sim\text{c}\sim\text{d a}}$ also starts from state $\{d, b, c, a\}$ and ends at state $\{d, c, b, a\}$, and it is already proven that sub-chain $\boxed{\text{b}\sim\text{c}\sim\text{d a}}$ is time reversible.

A similar argument can be given for every state in $\boxed{\text{a} \sim \text{b} \sim \text{c d}}$. Therefore, the Markov chain for $n = 4$ is time reversible.

*3. Induction hypothesis.*

Assuming that the Markov chain for $k$ elements $\{R_1, \ldots, R_k\}$ is time reversible, we shall prove that the Markov chain for $(k+1)$ elements $\{R_1, \ldots, R_k, R_{k+1}\}$ is time reversible.

Note that the Markov chain with $(k+1)$ elements will have $(k+1)$ sub-chains which are all symmetric and are all actually the same as the Markov chain for $k$ elements.

As before, take any state $\pi_s$, say $\pi_s = \{R_1, \ldots, R_{j-1}, R_j, R_{j+1}, \ldots, R_k, R_{k+1}\}$, from one of the sub-chains $\boxed{R_1 \sim \ldots \sim R_k \; R_{k+1}}$, and assume that the parent of $R_{k+1}$ is $R_j$. We need to prove that starting in $\pi_s$, any path back to $\pi_s$ has the same probability as the reversed path. Again, by our hypothesis, we don't need to consider paths that are entirely inside the sub-chain, but only paths that go *outside* the sub-chain. To reach any state outside the sub-chain, element $R_{k+1}$ has to be accessed. This makes the transition of the chain from $\pi_s$ to $\{R_1, \ldots, R_{j-1}, R_{k+1}, R_{j+1}, \ldots, R_k, R_j\}$ in sub-chain $\boxed{R_1 \sim \ldots \sim R_{j-1} \sim R_{j+1} \ldots \sim R_{k+1} \; R_j}$. Since all sub-chains are symmetric, to get back to $\pi_s$, we only need to consider the path that is directly from sub-chain $\boxed{R_1 \sim \ldots \sim R_{j-1} \sim R_{j+1} \ldots \sim R_{k+1} \; R_j}$. Assuming that records $R_i$ and $R_{i_p}$ are with parent/child relationship, a shortest path which visits only two states inside $\boxed{R_1 \sim \ldots \sim R_{j-1} \sim R_{j+1} \ldots \sim R_{k+1} \; R_j}$. is

$$
\begin{aligned}
& \{R_1, \ldots, R_{i_p}, \ldots, R_i, \ldots, R_j, \ldots, R_{k+1}\} \\
\rightleftharpoons \; & \{R_1, \ldots, R_{i_p}, \ldots, R_i, \ldots, R_{k+1}, \ldots, R_j\} \\
\rightleftharpoons \; & \{R_1, \ldots, R_i, \ldots, R_{i_p}, \ldots, R_{k+1}, \ldots, R_j\} \\
\rightleftharpoons \; & \{R_1, \ldots, R_i, \ldots, R_{i_p}, \ldots, R_j, \ldots, R_{k+1}\} \\
\rightleftharpoons \; & \{R_1, \ldots, R_{i_p}, \ldots, R_i, \ldots, R_j, \ldots, R_{k+1}\} .
\end{aligned}
$$

The product of the transition probabilities in the forward direction which is $s_{k+1} s_i s_j s_{i_p}$ equals to that in the reverse direction which is $s_i s_{k+1} s_{i_p} s_j$. The argument holds for any large path for the same reason stated for the case where $n = 4$. Therefore, the Markov chain is time reversible, and the inductive step is complete. The result follows. □

It directly leads to the equation shown in Corollary 3.1.

**Corollary 3.1** *Under the swap-with-parent-in-an-ss_tree heuristic, the stationary probabilities obey*

$$
\frac{P\{R_{i_1} \ldots \mathbf{R_{parent(i_j)}} \ldots \mathbf{R_{i_j}} \ldots R_{i_n}\}}{P\{R_{i_1} \ldots \mathbf{R_{i_j}} \ldots \mathbf{R_{parent(i_j)}} \ldots R_{i_n}\}} = \frac{s_{\mathbf{parent(i_j)}}}{s_{\mathbf{i_j}}} \qquad (4)
$$

*where* parent$(i_j)$ *is the index of the parent of* $R_{i_j}$ *in the corresponding ss_tree.* □

**Corollary 3.2** *Under the swap-with-parent-in-an-ss_tree heuristic the stationary probabilities obey:*

$$
\frac{P\{\ldots, \mathbf{R_i}, R_{i_1}, \ldots, R_{i_k}, \mathbf{R_j}, \ldots\}}{P\{\ldots, \mathbf{R_j}, R_{i_1}, \ldots, R_{i_k}, \mathbf{R_i}, \ldots\}} = \left(\frac{s_{\mathbf{i}}}{s_{\mathbf{j}}}\right)^{k_{ij}} \qquad (5)
$$

*where* $k_{ij} = |\, depth\,(\pi(i)) - depth\,(\pi(j))\,|$ *(in the corresponding ss_tree) and* $1 \le i, j \le n$ *if* $s_j \neq 0$.

**Proof:** This follows easily by successive swapping on the corresponding ss_tree using Equation (4). □

**Corollary 3.3** *Under the swap-with-parent-in-an-ss_tree heuristic, the stationary probabilities between any two arbitrary states* $\pi_s$ *and* $\pi_t$ *obey:*

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \prod_{1 \le i \le n} s_i^{\xi_i(\pi_t, \pi_s)} \tag{6}$$

*where* $\xi_i(\pi_t, \pi_s) = depth\,(\pi_t(i)) - depth\,(\pi_s(i))$ *for* $1 \le i \le n$ *is the difference between the depths of* $R_i$ *in ordering* $\pi_s$ *and* $\pi_t$. □

See [5, 15] for the proof of the above corollary.

Using the above we can present the expression for the asymptotic search cost of the SWPSST heuristic. It is a generalization of the one for the transposition rule and the swap-with-parent rule [5, 15, 19].

**Theorem 3.2** *Let* $\pi_0$ *denote the identity permutation on* $n$ *records* $\pi_0(i) = i$ *for* $1 \le i \le n$, *which is the optimal ordering under our assumption that* $s_i \ge s_{i+1}$ *for* $1 \le i \le n$. *Let* $\xi_i(\pi_0, \pi)$ *denote the quantity* $(depth\,(i) - depth\,(\pi(i)))$ *for any permutation* $\pi$ *and* $1 \le i \le n$, *which is the difference between the depths of* $R_i$ *in the optimal ordering and in ordering* $\pi$ *of the corresponding ss_tree. Then the asymptotic search cost for the swap-with-parent-in-an-ss_tree heuristic is*

$$P\{\pi_0\} \sum_{all\ \pi} \left( \prod_{1 \le i \le n} s_i^{\xi_i(\pi_0, \pi)} \sum_{1 \le j \le n} s_j \pi(j) \right) \tag{7}$$

*where*

$$P\{\pi_0\} = \left( \sum_{all\ \pi} \prod_{1 \le i \le n} s_i^{\xi_i(\pi_0, \pi)} \right)^{-1}. \tag{8}$$

**Proof:** The average search cost of an algorithm can be calculated by

$$\sum_{\text{all } \pi} \left( P\{\pi\} \sum_{1 \le j \le n} s_j\, \pi(j) \right).$$

Using Corollary 3.3, we have,

$$P\{\pi\} = P\{\pi_0\} \prod_{1 \le i \le n} s_i^{\xi_i(\pi_0, \pi)} \qquad \text{for all } \pi. \tag{9}$$

Consequently

Average search cost for SWPSST

$$= \sum_{\text{all } \pi} \left( \left( P\{\pi_0\} \prod_{1 \le i \le n} s_i^{\xi_i(\pi_0, \pi)} \right) \sum_{1 \le j \le n} s_j\, \pi(j) \right)$$

$$= P\{\pi_0\} \sum_{\text{all } \pi} \left( \left( \prod_{1 \le i \le n} s_i^{\xi_i(\pi_0, \pi)} \right) \sum_{1 \le j \le n} s_j \pi(j) \right),$$

thus Equation (7) is proved.
Since

$$\sum_{\text{all } \pi} P\{\pi\} = 1,$$

applying Equation (9), we have

$$P\{\pi_0\} \sum_{\text{all } \pi} \left( \prod_{1 \le i \le n} s_i^{\xi_i(\pi_0, \pi)} \right) = 1,$$

which proves Equation (8). □

Therefore, to find the cost of the SWPSST rule with a specific ss_tree, the only thing we need to do is to find the value of $\xi_i(\pi_0, \pi)$ and replace it in Equation (7) and (8). For example, the value of $\xi_i(\pi_0, \pi)$ for the 3-branch tree shown in Figure 2 (d2) is

$$\lceil \frac{\pi_0(i) - 1}{3} \rceil - \lceil \frac{\pi(i) - 1}{3} \rceil .$$

Similar straightforward expressions can be written for each of the ss_tree structures given in Figure 2 except (f) and (g). They are omitted in the interest of brevity.

Using time reversibility, we can prove the following results. The proof of the results is very close to the proof of their analogous theorems for the swap-with-parent heuristic [5, 15]. Therefore, in the interest of brevity, we will not show the proofs here.

**Theorem 3.3** *The swap-with-parent-in-an-ss_tree heuristic is expedient since*

$$P\{R_j \text{ precedes } R_i\}_{SWPSST} > \frac{1}{2} \qquad \text{if } s_j > s_i .$$

□

**Theorem 3.4** *For any two records $R_i$ and $R_j$, let $k_{ij} = |\, depth\,(\pi(i)) - depth\,(\pi(j))\,|$ in the corresponding ss_tree. Then under the swap-with-parent-in-an-ss_tree heuristic we have:*

$$P\{R_j \text{ precedes } R_i \mid k_{ij} = d\} = \frac{1}{1 + \left(\frac{s_i}{s_j}\right)^d} = \frac{s_j^d}{s_i^d + s_j^d}. \tag{10}$$

□

The performance of the SWPSST heuristic (both asymptotic search cost and convergence) depends heavily on the type of the underlying ss_tree. It is not difficult to see that the rule will not cost less than the transposition rule when the corresponding ss_tree has more than one branch. However we conjecture that it costs less than the move-to-front rule. Furthermore, the smaller the number of records in the leaf level in the corresponding ss_tree, the better the algorithm will be. Equivalently, the bushier (more branches) the tree is, the worse the algorithm will be. This is because to make the algorithm better than the move-to-front rule, we would like that

$$P\{R_j \text{ precedes } R_i\}_{SWPSST} > P\{R_j \text{ precedes } R_i\}_{MTF} = \frac{s_j}{s_i + s_j}$$

given that $s_j > s_i$ for $1 \leq i, j \leq n$ and $j \neq i$.

From Theorem 3.4 above, we know that if $s_j < s_i$, then

$$P\{R_j \text{ precedes } R_i \mid k_{ij} = d = 0\}_{SWPSST} < \frac{s_j}{s_i + s_j}. \tag{11}$$

$$P\{R_j \text{ precedes } R_i \mid k_{ij} = d \geq 1\}_{SWPSST} \geq \frac{s_j}{s_i + s_j}. \tag{12}$$

Therefore we would like the *asymptotic* probability of two records being in the same level in the corresponding ss_tree to be smaller than that of them *not* being in the same level. In other words, the smaller the number of branches the tree has, the less the algorithm costs.

The best-case performance of the algorithm happens when the underlying ss_tree is an *unary* tree, in which case the algorithm is equivalent to the transposition rule. The worst-case occurs when the underlying ss_tree has $(n-1)$ branches, i.e., the first record is the parent of all other records in the list. However, the speed of convergence of the algorithm will be more than that of the transposition.

Let us revisit some of the examples of the various ss_trees are shown from Section 3.1 (see also Figure 2). The *k-branch ss_tree* shown in Example 3 (Figure 2 (c)) when $k$ is chosen properly, and the ss_tree shown in Example 6 (Figure 2 (f)) will be better choices for the underlying ss_tree when using the swap-with-parent-in-an-ss_tree heuristic. The ss_trees shown in Example 4 and 5 (Figure 2 (d1), (d2), (d3) and (e)) are the bad choices for the reasons listed above.

### 3.3 Drawbacks of the SWPSST Rule and its Modification - the Move-to-Parent-in-an-SS_Tree Rule

The drawback of the swap-with-parent heuristic persists in the SWPSST heuristic. The scenario is exactly the same - the elements with zero (or negligible) access probabilities will prove to be a drag to the entire access strategy. After those elements eventually fall to the leaf level, they will not be moved again, therefore the cost of future accesses to the elements *behind* them in the list will be increased, and the situation will not get better. This is again the reason why we prefer the ss_tree not to be 'bushy'.

The solution to this problem is the same as the solution to the swap-with-parent heuristic. Instead of swapping the accessed record with its parent in the associated ss_tree and leaving all the records in between unchanged, we will move the accessed record to its parent's position and shift its parent and all the records in between back one position. We shall call this scheme the *move-to-parent-in-an-ss_tree* (MTPSST) heuristic.

Notice that if the underlying ss_tree is an *k-branch ss_tree* (see Example 3 in Section 3.1 and Figure 2 (c)), the corresponding MTPSST is the *move-ahead-k* scheme. If the ss_tree is an $(n-1)$-branch tree, the scheme is the *move-to-front* scheme, and if the ss_tree is a unary ss_tree (one-branch tree), the scheme is the transposition scheme.

We conjecture that its average search cost is lower than the move-to-front rule and higher than the transposition rule. Absolute analysis of it is needed to allow realistic comparisons. Also, like the move-to-parent heuristic, this scheme avoids the drawbacks of many other algorithms including the drawbacks related to 'locality' mentioned in [5, 8, 15] as is explained below.

## 4 The MTPSST Heuristics and Locality

In self-organizing sequential search algorithms, it is commonly assumed that the accesses to the record in the list are made independent of previous accesses and that the access probabilities do

not vary with respect to time, (stationary). Consequently, the majority of the analyses on self-organizing sequential search algorithms assume that all access sequences which have the same set of access probabilities are equally likely.

This assumption does not take into account a common attribute of access sequences called *locality* which makes the assumption unreasonable in many applications. Locality means that some "subsequences of the entire access sequence may have relative access frequencies that are drastically different from the overall relative access frequencies" [8].

Hester and Hirschberg [8] showed how the *locality* of the access sequence affects the average search cost of a scheme through the following example. Consider a list of 26 records which are the 26 English letters $\{a, b, \ldots, z\}$ in a random order. Each record is accessed exactly 10 times, that is, $s_a = s_b = \ldots = s_z = 1/26$. Let us assume that the reordering rule is the *move-to-front.*

If the access sequence is as below,

$$\underbrace{\underbrace{a, \ldots, z}, \underbrace{a, \ldots, z}, \ \ldots, \ \underbrace{a, \ldots, z}}_{10 \ times} \tag{13}$$

then under the move-to-front rule, accesses to each record (except the first access of each record) will always take 26 probes. Therefore the total number of probes required to access the records in the input sequence (except the first access to each record) is $9 \times 26 \times 26$, and the first access takes between 26 and the position of the record in the list. The best-case scenario happens when the list is initially in alphabetical order. Then the cost of the move-to-front rule will be:

$$\frac{9 \times 26 \times 26 + \sum_{i=1}^{26} i}{260} = 24.75$$

Now consider another access sequence which has the same probability distribution as the previous, but are in a different order shown below.

$$\underbrace{a, \ldots, a}_{10 \ times}, \underbrace{b, \ldots, b}_{10 \ times}, \ \ldots, \ \underbrace{z, \ldots, z}_{10 \ times} \tag{14}$$

In this case, all accesses (except the first) to each record will only take one probe. The worst-case scenario happens when the list is initially in the reversed alphabetical order, and the cost of the move-to-front rule is:

$$\frac{9 \times 26 \times 1 + \sum_{i=1}^{26} i}{260} = 3.5$$

Note that the worst-case cost of the second scenario is far less than the best-case cost of the first scenario. This indicates that the cost of a scheme can differ greatly for the same fixed probability distribution under the move-to-front rule when the order of the actual input sequences are different.

However, when the underlying ss_tree is not the $(n-1)$-branch tree, the MTPSST rules will not have the above problem since the rules will not be as drastic as the move-to-front rule when moving a record forward. It can be easily verified that the cost of the first input sequence in the above example under any of the MTPSST rules (except when the corresponding ss_tree is the $(n-1)$-branch tree) is less than that of the move-to-front rule. Simultaneously, the cost of the second sequence in the above example will be more than that of the move-to-front rule. Therefore for the same fixed probability distribution, when the order of the actual input sequences are different, the cost of the MTPSST rules (when the corresponding ss_tree is not the $(n-1)$-branch tree) will not differ as much as the move-to-front rule. This demonstrates that such MTPSST rules respond in a superior fashion when it encounters this kind of 'locality'.

Another case that can happen when using the transposition rule and the swap-with-parent rule is when two records are alternately accessed many times, they will continue exchanging places without advancing toward the front of the list. However, the MTPSST rules will not have this problem.

# 5 Conclusion

In this paper, we have formally defined the *tree transformation process* and introduced a generalization of the transposition rule and the swap-with-parent rule - the *swap-with-parent-in-an-ss_tree* (SWPSST) heuristic and its modification - the *move-to-parent-in-an-ss_tree* (MTPSST) heuristic. Both of these are defined on the underlying implicit tree structure defined on the list - a structure we have called the sequential search tree or ss_tree. We have shown the time reversibility of the SWPSST schemes and a variety of results which specialize to the analogous results for transposition rule and the swap-with-parent rule.

The performance of both of the schemes depends heavily on the type of the underlying ss_tree used. In both of the schemes, we believe that in terms of accuracy, the best-case scenario happens if the underlying ss_tree is the unary tree, i.e., the tree with one branch, in which case the scheme is actually the transposition rule. This heuristic is the most sluggish scheme in terms of convergence. The worst-case scenario occurs if the underlying ss_tree is an $(n-1)$-branch tree. Generally speaking, the 'bushier' the ss_tree is, the more the algorithms costs, and the faster the convergence rate is.

We conjecture that SWPSST heuristic costs less than the move-to-front heuristic and that the MTPSST heuristic is even better. Further analyses for both of the schemes are needed. This indeed gives rise to numerous open problems.

# Appendix

# A Summary of the Swap-with-Parent Scheme

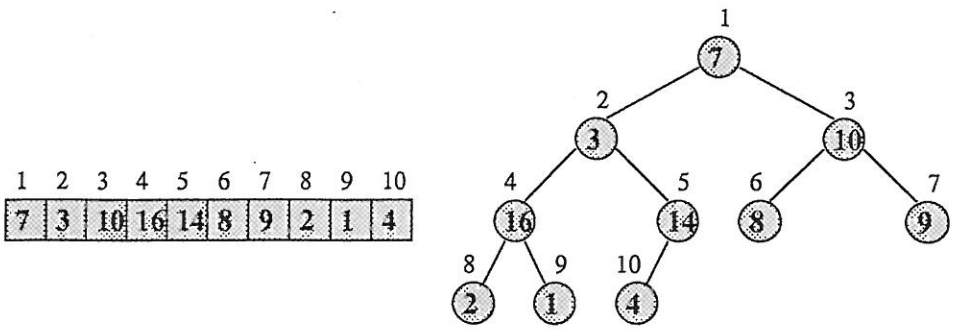
Throughout this paper we have alluded to our recently designed algorithm, the swap-with-parent (SWP) heuristic. However as this is yet unpublished, we have included the key results concerning the scheme in this Appendix to assist the reader. [1]

For any given list of records $\{R_1, R_2, \ldots, R_n\}$, we can conceptually construct a *heap structure* with *no* ordering constraints between parents and their children. For example, given a list of elements $\{7, 3, 10, 16, 14, 8, 9, 2, 1, 4\}$, the corresponding "heap" structure with *no ordering constraints* is shown in Figure 4. We shall refer to this structure as a *sequential_heap* or *s_heap.*

Suppose we have a set of $n$ records $\{R_1, R_2, \ldots, R_n\}$ which are in an arbitrary order $\pi$, so that $R_i$ is in position $\pi(i)$ for $1 \leq i \leq n$. Using the *swap-with-parent* heuristic, whenever a record $R_i$ is found in position $\pi(i)$, the list is rearranged by exchanging the positions of $R_i$ and its parent which is in position $\lfloor \pi(i) / 2 \rfloor$ and leaving all the other records untouched; if $R_i$ heads the list nothing is done.

By way of example, consider a list of 10 elements $\{7, 3, 10, 16, 14, 8, 9, 2, 1, 4\}$, using the swap-with-parent rule, Figure 5 shows the changes on the orders of the list after elements 16 and 1 are accessed consecutively.

[1] At the discretion of the editor, we are willing to delete this Appendix in the final manuscript.




( a ) A list of 10 elements ( b ) The corresponding heap

Figure 4: A list viewed as a "heap".

The major results of the swap-with-parent rule are shown in the following theorems and corollaries. All the proofs can be found in [5, 15].

**Theorem A.1** *The Markov chain which results from using the swap-with-parent heuristic is time reversible.* □

**Theorem A.2** *For a given set of records* $\{R_1, R_2, \ldots, R_n\}$ *with respective access probabilities* $\{s_1, s_2, \ldots, s_n\}$, *under the swap-with-parent heuristic the stationary probabilities obey:*

$$\frac{P\{R_{i_1} \ldots \mathbf{R_{i_j}} \ldots \mathbf{R_{i_{2j}}} \ldots R_{i_n}\}}{P\{R_{i_1} \ldots \mathbf{R_{i_{2j}}} \ldots \mathbf{R_{i_j}} \ldots R_{i_n}\}} = \frac{s_{\mathbf{i_j}}}{s_{\mathbf{i_{2j}}}} \tag{15}$$

*and*

$$\frac{P\{R_{i_1} \ldots \mathbf{R_{i_j}} \ldots \mathbf{R_{i_{2j+1}}} \ldots R_{i_n}\}}{P\{R_{i_1} \ldots \mathbf{R_{i_{2j+1}}} \ldots \mathbf{R_{i_j}} \ldots R_{i_n}\}} = \frac{s_{\mathbf{i_j}}}{s_{\mathbf{i_{2j+1}}}} \tag{16}$$

*for* $1 \le j < n$ *if* $s_k \neq 0$ *for* $1 \le k \le n$. □

**Corollary A.1** *Under the swap-with-parent rule the stationary probabilities obey:*

$$\frac{P\{\ldots, \mathbf{R_i}, R_{i_1}, \ldots, R_{i_k}, \mathbf{R_j}, \ldots\}}{P\{\ldots, \mathbf{R_j}, R_{i_1}, \ldots, R_{i_k}, \mathbf{R_i}, \ldots\}} = \left(\frac{s_{\mathbf{i}}}{s_{\mathbf{j}}}\right)^{k_{ij}} \tag{17}$$

*where* $k_{ij} = |\lfloor \lg(\pi(i)) \rfloor - \lfloor \lg(\pi(j)) \rfloor|$, *for* $1 \le i, j \le n$ *if* $s_j \neq 0$. □

It is advantageous to see the results shown above through an example. Considering a list of 10 elements $\{R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9, R_{10}\}$. Figure 6 illustrates the sequence of transformations from state $\pi_s$ to state $\pi_t$, where $\pi_s = \{R_1, R_2, \mathbf{R_3}, R_4, \ldots, R_8, \mathbf{R_9}, R_{10}\}$ and $\pi_t = \{R_1, R_2, \mathbf{R_9}, R_4, \ldots, R_8, \mathbf{R_3}, R_{10}\}$.

To go from state $\pi_s$ to state $\pi_t$, swapping element $R_9$ with its parent until it reaches the root and then swapping $R_3$ with it, now $R_9$ is in the position of $R_3$. Then swapping $R_1$ with $R_3$, then $R_2$ with $R_3$, then $R_4$ with $R_3$. We have reached state $\pi_t$. Let $\pi_i$ denote all the states

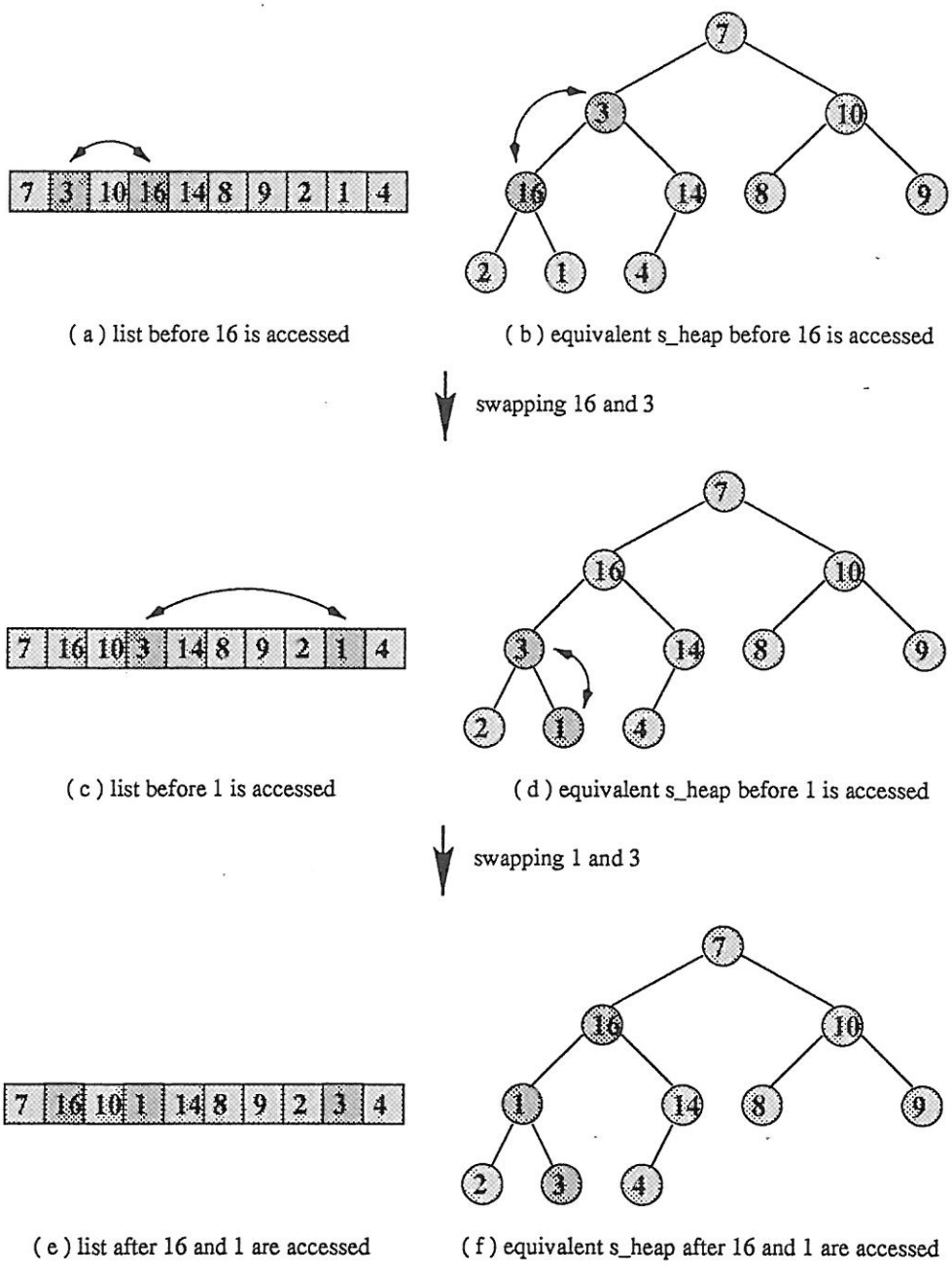



Figure 5: Operations of swapping with parent.

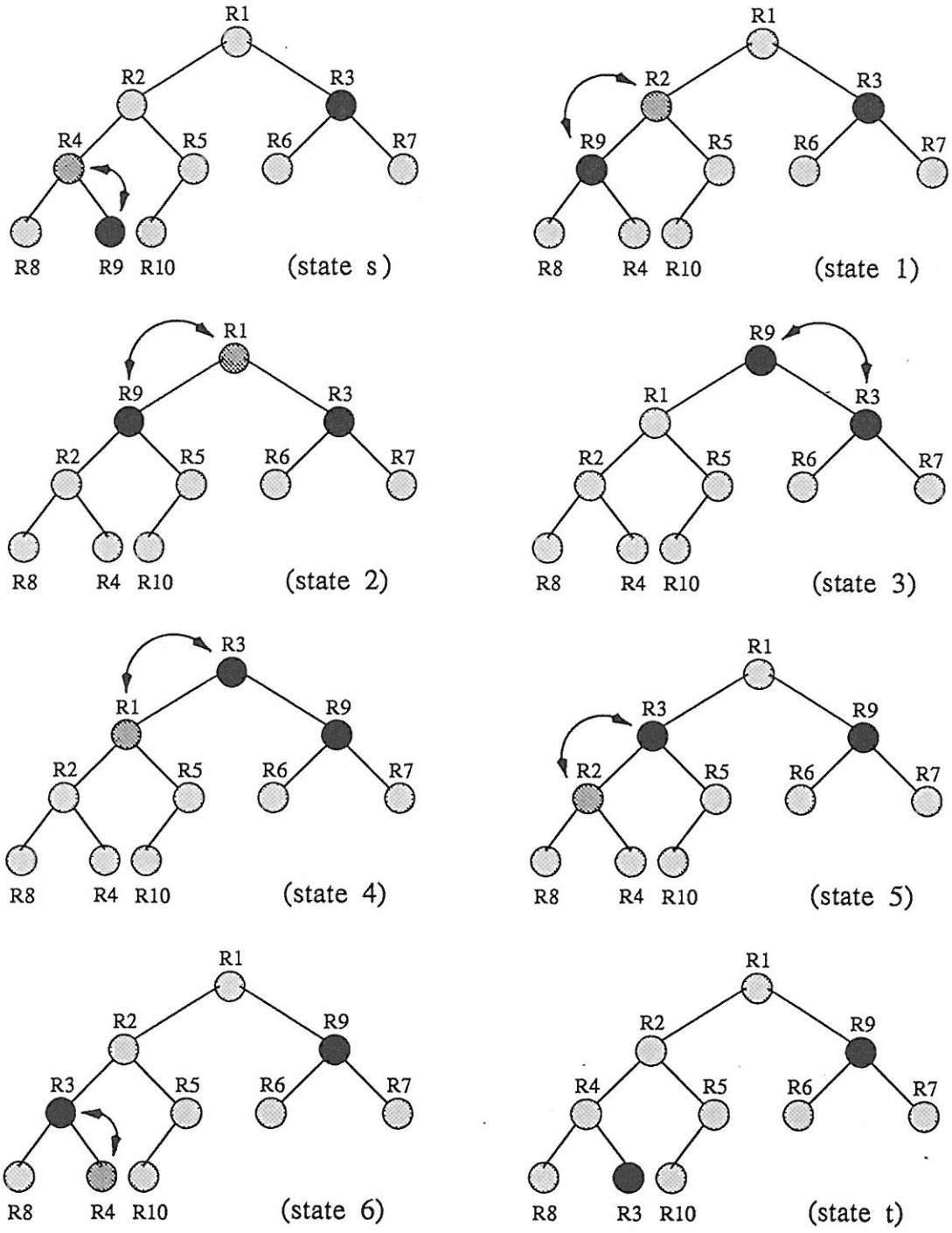



Figure 6: Transformations from state $\pi_s$ to $\pi_t$ through $\pi_1, \ldots, \pi_6$.

during the transformation, then

$$\begin{aligned}
\pi_s &= \{R_1, R_2, \mathbf{R_3}, R_4, R_5, R_6, R_7, R_8, \mathbf{R_9}, R_{10}\}, \\
\pi_1 &= \{R_1, R_2, R_3, \mathbf{R_9}, R_5, R_6, R_7, R_8, \mathbf{R_4}, R_{10}\}, \\
\pi_2 &= \{R_1, \mathbf{R_9}, R_3, \mathbf{R_2}, R_5, R_6, R_7, R_8, R_4, R_{10}\}, \\
\pi_3 &= \{\mathbf{R_9}, \mathbf{R_1}, R_3, R_2, R_5, R_6, R_7, R_8, R_4, R_{10}\}, \\
\pi_4 &= \{\mathbf{R_3}, R_1, \mathbf{R_9}, R_2, R_5, R_6, R_7, R_8, R_4, R_{10}\}, \\
\pi_5 &= \{\mathbf{R_1}, \mathbf{R_3}, R_9, R_2, R_5, R_6, R_7, R_8, R_4, R_{10}\}, \\
\pi_6 &= \{R_1, \mathbf{R_2}, R_9, \mathbf{R_3}, R_5, R_6, R_7, R_8, R_4, R_{10}\}, \\
\pi_t &= \{R_1, R_2, \mathbf{R_9}, R_4, R_5, R_6, R_7, R_8, \mathbf{R_3}, R_{10}\}.
\end{aligned}$$

From Equation (15), we have

$$\begin{aligned}
P\{\pi_s\} &= (s_4 / s_9)\, P\{\pi_1\}, \\
P\{\pi_1\} &= (s_2 / s_9)\, P\{\pi_2\}, \\
P\{\pi_2\} &= (s_1 / s_9)\, P\{\pi_3\}, \\
P\{\pi_3\} &= (s_9 / s_3)\, P\{\pi_4\}, \\
P\{\pi_4\} &= (s_3 / s_1)\, P\{\pi_5\}, \\
P\{\pi_5\} &= (s_3 / s_2)\, P\{\pi_6\}, \\
P\{\pi_6\} &= (s_3 / s_4)\, P\{\pi_t\}.
\end{aligned}$$

Therefore,

$$P\{\pi_s\} = \frac{s_4}{s_9} \times \frac{s_2}{s_9} \times \frac{s_1}{s_9} \times \frac{s_9}{s_3} \times \frac{s_3}{s_1} \times \frac{s_3}{s_2} \times \frac{s_3}{s_4} \times P\{\pi_t\} = \left(\frac{s_3}{s_9}\right)^2 P\{\pi_t\}. \tag{18}$$

Since $depth(R_3) = 1$ and $depth(R_9) = 3$, the difference between the depths of $R_3$ and $R_9$ is 2. Therefore Equation (18) above verifies the result shown in Corollary A.1.

**Theorem A.3** *Under the swap-with-parent heuristic, the stationary probabilities between any two arbitrary states $\pi_s$ and $\pi_t$ obey:*

$$\frac{P\{\pi_s\}}{P\{\pi_t\}} = \prod_{1 \le i \le n} s_i^{\xi_i(\pi_t, \pi_s)} \tag{19}$$

*where $\xi_i(\pi_t, \pi_s) = \lfloor \lg(\pi_t(i)) \rfloor - \lfloor \lg(\pi_s(i)) \rfloor$ for $1 \le i \le n$ is the difference between the depths of $R_i$ in ordering $\pi_s$ and $\pi_t$.* □

Rivest gave an expression for the asymptotic search cost of the transposition rule (see [19]). The analogous result for the average search cost of the swap-with-parent rule is given below.

**Theorem A.4** *Let $\pi_0$ denote the identity permutation on $n$ records $\pi_0(i) = i$ for $1 \le i \le n$, which is the optimal ordering under our assumption that $s_i \ge s_{i+1}$ for $1 \le i \le n$. Let $\xi_i(\pi_0, \pi)$ denote the quantity $(\lfloor \lg(i) \rfloor - \lfloor \lg(\pi(i)) \rfloor)$ for any permutation $\pi$ and $1 \le i \le n$, which is the difference between the depths of $R_i$ in the optimal ordering and in ordering $\pi$. Then the asymptotic search cost for the swap-with-parent heuristic is*

$$P\{\pi_0\} \sum_{all\ \pi} \left( \prod_{1 \le i \le n} s_i^{\xi_i(\pi_0, \pi)} \sum_{1 \le j \le n} s_j \pi(j) \right) \tag{20}$$

*where*

$$P\{\pi_0\} = \left( \sum_{all\ \pi} \prod_{1 \le i \le n} s_i^{\xi_i(\pi_0,\pi)} \right)^{-1} . \tag{21}$$

□

Using time reversibility, we can now prove the following results.

**Theorem A.5** *The swap-with-parent heuristic is expedient since*

$$P\{R_j \text{ precedes } R_i\}_{SWP} > \frac{1}{2} \qquad \text{if } s_j > s_i .$$

□

Other results for the scheme can be found in [5, 15].

## Acknowledgments

We are grateful to Jason Morrison for taking the time to discuss the problem with us which provided us with very helpful and motivational comments.

## References

[1] E. J. Anderson, P. Nash & R. R. Weber (1982). A counterexample to a conjecture on optimal list ordering. *J. Appl. Prob. 19, 3*, pp.730-732.

[2] J. L. Bentley & C. C. (1985). Amortized analyses of self-organizing sequential search heuristics. *Proc. 20th Allerton Conference on Comm. Control, and Computing.*

[3] J. R. Bitner (1976). Heuristics that dynamically alter data structures to reduce their access time. *University of Illinois Report UIUCDCS-R-76-818*, July, 1976 (Ph.D. thesis).

[4] J. R. Bitner (1979). Heuristics that dynamically organize data structures. *SIAM J. Computing, 8, 1*, pp. 82-110.

[5] J. Dong (1997). Time reversible self-organizing sequential search algorithms. M. Sc. thesis. *School of computer science, Carleton University.*

[6] G. H. Gonnet, J. I. Munro & H. Suwanda (1981). Exegesis of self-organizing linear search. *SIAM J. Computing, 10, 3*, pp. 613-637.

[7] W. J. Hendricks (1973). An extension of a theorem concerning an interesting Markov chain. *J. Appl. Prob. 10*, pp886-890.

[8] J. H. Hester & D. S. Hirschberg (1985). Self-organizing linear search. *ACM Computing Surveys*, pp. 295-311.

[9] Y. C. Kan & S. M. Ross (1980). Optimal list order under partial memory constraints. *J. Appl. Prob. 17*, pp1004-1015.

[10] F. P. Kelly (1987). *Reversibility and Stochastic Networks.* John Wiley & Sons, Chichester.

[11] S. Karlin & H. M. Taylor (1975). *A First Course in Stochastic Processes.* Academic Press, New York.

[12] D. E. Knuth (1973). *The Art of Computer Programming,* Vol. 3, Addison-Wesley, Reading, Ma..

[13] J. McCabe (1965). On serial files with relocatable records. *Operations Research,* 12, pp. 609-618.

[14] B. J. Oommen & J. Dong (1997). On the time reversibility of a well known self-organizing sequential search algorithm. *In preparation.*

[15] B. J. Oommen & J. Dong (1997). The swap-with-parent scheme: a self-organizing sequential search algorithm which uses non-lexicographic heaps. *In preparation.*

[16] B. J. Oommen & E. R. Hansen (1987). List organizing strategies using stochastic move-to-front and stochastic move-to-rear operations. *SIAM J. Computing, 16*, pp. 705-716.

[17] B. J. Oommen, E. R. Hansen & J. I. Munro (1990). Deterministic optimal and expedient move-to-rear list organizing strategies. *Theoretical Comp. Sci., 74*, pp. 183-197.

[18] B. J. Oommen & D. T. H. Ng (1993). An optimal absorbing list organization strategy with constant memory requirements. *Theoretical Comp. Sci., 119*, pp. 355-361.

[19] R. L. Rivest (1976). On self-organizing sequential search heuristics. *Comm. ACM, 19,* pp. 63-67.

[20] S. M. Ross (1980). *Introduction to Probability Models*, 2nd edition. Academic Press, London.

[21] A. M. Tenenbaum & R. M. Nemes (1982). Two spectra of self-organizing sequential search algorithms. *SIAM J. Computing, 11,* pp. 557-566.

## School of Computer Science, Carleton University

## Recent Technical Reports

**TR-96-01** **Hierarchical Load Sharing Policies for Distributed Systems**
Sivarama Dandamudi and Michael Lo, January 1996

**TR-96-02** **Randomized Parallel List Ranking for Distributed Memory Multiprocessors**
Frank Dehne and Siang W. Song, January 1996

**TR-96-03** **Randomized Sorting on Optically Interconnected Parallel Computer**
G. Bhattacharya, J. Chrostowski, F. Dehne, P. Palacharla, January 1996

**TR-96-04** **Authenticated Multi-Party Key Agreement**
Mike Just and Serge Vaudenay, February 1996

**TR-96-05** **Boolean Routing on Cayley Networks**
Evangelos Kranakis and Danny Krizanc, February 1996

**TR-96-06** **Secure Non-Interactive Electronic Cash**
Patrick Morin, February 1996

**TR-96-07** **An Upper Bound for a Basic Hypergeometric Series**
Lefteris M. Kirousis, Evangelos Kranakis and Danny Krizanc, March 1996

**TR-96-08** **A Formal Theory for Optimal and Information Theoretic Syntactic Pattern Recognition**
B.J. Oommen and R.L. Kashyap, March 1996

**TR-96-09** **A Better Upper Bound for the Unsatisfiability Threshold**
Lefteris M. Kirousis, Evangelos Kranakis and Danny Krizanc, March 1996

**TR-96-10** **Minimal Sense of Direction in Regular Networks**
Paola Flocchini, March 1996

**TR-96-11** **Correlation Inequality and its application to a Word Problem**
Dimitris Achlioptas, Lefteris M. Kirousis, Evangelos Kranakis, Danny Krizanc, Michael S.O. Molloy, March 1996

**TR-96-12** **On Systems with Sense of Direction**
Paola Flocchini, Alessandro Roncato, Nicola Santoro, April 1996

**TR-96-13** **Computing on Anonymous Networks with Sense of Direction**
Paola Flocchini, Alessandro Roncato, Nicola Santoro, April 1996

**TR-96-14** **Symmetries and Sense of Direction in Labeled Graphs**
Paola Flocchini, Alessandro Roncato, Nicola Santoro, April 1996

**TR-96-15** **Distance Routing on Series Parallel Networks**
Paola Flocchini, Flaminia L. Luccio, April 1996

**TR-96-16** **Maximal Length Common Non-intersecting Paths**
Evangelos Kranakis, Danny Krizanc, Jorge Urrutia, May 1996

**TR-96-17** **Discrete Vector Quantization for Arbitrary Distance Function Estimation**
John Oommen, I. Kuban Altinel, Necati Aras, May 1996

**TR-96-18** **Symmetry and Computability in Anonymous Networks: A Brief Survey**
Evangelos Kranakis, July 1996

**TR-96-19** **Many-to-One Packet Routing via Matchings**
Danny Krizanc, Louxin Zhang, July 1996

**TR-96-20** **Power Consumption in Packet Radio Networks**
Lefteris M. Kirousis, Evangelos Kranakis, Danny Krizanc, Andrzej Pelc, August 1996

**TR-96-21** **Performance of Hierarchical Processor Scheduling in Shared-Memory Multiprocessor Systems**
Sivarama P. Dandamudi, Samir Ayachi, August 1996

**TR-96-22** **Performance of Hierarchical Load Sharing in Heterogeneous Distributed Systems**
Michael Lo and Sivarama P. Dandamudi, August 1996

**TR-96-23** **Performance Impact of I/O on Sender-Initiated and Receiver-Initiated Load Sharing Policies in Distributed Systems**
Sivarama P. Dandamudi and Hamid Hadavi, August 1996

**TR-96-24** **Characterization of Domino Tilings of Squares with Prescribed Number of Nonoverlapping 2x2 Squares**
Evangelos Kranakis, September 1996

**TR-96-25** **On the Impact of Sense of Direction on Communication Complexity**
Paola Flocchini, Bernard Mans, Nicola Santoro, October 1996

**TR-96-26** **The Effect of Scheduling Discipline on Dynamic Load Sharing in Heterogeneous Distributed Systems**
Sivarama P. Dandamudi, November 1996

**TR-96-27** **Approximating the Unsatisfiability Threshold of Random Formulas**
Lefteris M. Kirousis, Evangelos Kranakis, Danny Krizanc, Yannis C. Stamatiou, November 1996

**TR-96-28** **Paper foldings as chaotic dynamical systems**
F. Geurts, November 1996

**TR-96-29** **Compositional complexity in cellular automata: a case study**
P. Flocchini, F. Geurts, November 1996

**TR-96-30** **Compositional complexity in dynamical systems**
F. Geurts, November 1996

**TR-96-31** **Compositional experimental analysis of cellular automata: attraction properties and logic disjunction**
P. Flocchini, F. Geurts, N. Santoro, November 1996

**TR-96-32** **Approximating Weighted Shortest Paths on Polyhedral Surfaces**
Mark Lanthier, Anil Maheshwari and Jörg-Rüdiger Sack, December 1996

**TR-97-01** **Performance Comparison of Processor Scheduling Strategies in a Distributed-Memory Multicomputer System**
Yuet-Ning Chan, Sivarama P. Dandamudi and Shikharesh Majumdar, January 1997

**TR-97-02** **Performance Evaluation of a Two-Level Hierarchical Parallel Database System**
Yifeng Xu and Sivarama P. Dandamudi, January 1997

**TR-97-03** **Using Networks of Workstations for Database Query Operations**
Sivarama P. Dandamudi, January 1997

**TR-97-04** **A Performance Study of Locking Granularity in Shared-Nothing Parallel Database Systems**
S. Dandamudi, S.L. Au, and C.Y. Chow, January 1997

**TR-97-05** **Continuous Learning Automata Solutions to the Capacity Assignment Problem**
B. John Oommen and T. Dale Roberts, April 1997

**TR-97-06** **Discrete Learning Automata Solutions to the Capacity Assignment Problem**
B. John Oommen and T. Dale Roberts, May 1997

**TR-97-07** **The Swap-with-Parent Scheme: a Self-Organizing Sequential Search Algorithm which uses Non-Lexicographic Heaps**
John Oommen and Juan Dong, May 1997

**TR-97-08** **Time reversibility: a mathematical tool for creating arbitrary generalized swap-with-parent self-organizing lists**
John Oommen and Juan Dong, May 1997

**TR-97-09** **Designing Syntactic Pattern Classifiers Using Vector Quantization and Parametric String Editing**
B. J. Oommen, R.K.S. Loke, May 1997

**TR-97-10** **Performance of a Parallel Application on a Network of Workstations**
A. Piotrowski and S.P. Dandamudi, May 1997

All the above-listed reports are available electronically. For further information,
(1) Access the school's worldwide web server at http://www.scs.carleton.ca.
(2) Click on to the 'Technical Reports' entry found on the main home page.
(3) Make sure that you have set the 'download to local disk' option on your mosaic or other client program. Click onto the appropriate technical reports and the postscript file will be downloaded onto your local disk.